Isabelle/HOL: a Proof Assistant For Automated Testing of Haskell Algorithms

Juan C. Rey
law.smart.contract@gmail.com

Smart Contract Audit Token
www.scatdao.com

September 27, 2021

Abstract. This article describes Isabelle's functionality as a proof assistant for Haskell algorithms. Isabelle is a general purpose test assistant that supports various theories and notations. The most representative notation is HOL (High Order Logic) and its syntax will be familiar to programmers who use functional programming languages. The research focuses on the functionality of the Isabelle to perform automated tests, counterexample search, and formal verification of Haskell algorithms. Ideal for searching for hidden behaviors of an algorithm, since bugs or errors can become the vulnerabilities of a system in terms of security. This article will also discuss the syntactic characteristics between Haskell and Isabelle, as well the transcription of signs and operators. The methodology we follow is the systematic review of software engineering research proposed by Kitchenham. The literature from 2010 to 2021 is systematically analyzed in different search engines.

1 Introduction

The functional programming paradigm facilitates the formal verification of our software programs. This allows programmers to be certain about their behavior at runtime and the correct formation of the arguments, thus avoiding errors that can result in serious consequences, especially if the algorithms belong to systems that must remain secure and trustworthy, which will be achieved if they know all the possibilities in the universe of cases of how a program could behave.

The formal verification of medium and large projects through manual methods, would result in a considerable use of time and resources. Luckily, there currently are alternatives in this field, which can help to obtain the results we want in a faster way with a higher degree of precision, through the running automated tests and using different types of notation for documentation.

2 Methodology

The methodology that we follow, is the systematic review of research in software engineering proposed by Kitchenham[1] The most relevant literature was compiled according to the title and abstract, making use of database search engines, Google Scholar, IOHK research , ACM, IEEE, Web of Science, Springer, IACR ePrint and arXiv, because they are repositories that store documents related to software engineering.

Our research focuses on Isabelle, its base theory, characteristics, types of notation and level of documentation, within the functional programming scheme, so we only include the works, which are identified with the keyword research categories, "functional programming", "formal verification", "proof assistant", also includes the information extracted from the official documentation found in the website databases.

Regarding the classification of the study object and the type of study of articles, they were read individually identifying the sections that accompany the object and the focus study of the present investigation, instead of assigning an identifier for each item, however, if two articles referred to the same type of study, it was avoided falling into iterations, but still utilizing what could be complementary.

3 Functional Programming Paradigm

In computing there are different models on how to approach problem solving. They can be recognized as programming paradigms and each can be found in different types of language. Some of a specific domain and others that mix several characteristics of multiple paradigms, among which we can find the Imperative, the Object-Oriented, the Functional Paradigm, and the Logical Paradigm, as the main ones.

The functional paradigm opts for embodying the theory of mathematical functions. It allows the programmer to play at a more fundamental and abstract level. This has its benefits when programming, since certain low-level characteristics are delegated to the machine.

In a functional programming language, you can identify the following components; data objects, built-in functions, and functional forms, as higher-order functions that can be used to build a new function. On the other hand, the secondary effects are eliminated or reduced and the functions are used to generate new values, it is declarative and it is not procedural. Languages like LISP, ML, Scheme, Miranda, and Haskell support this paradigm.

[1]Kitchenham, Barbara. (2012). Systematic review in software engineering: where we are and where we should be going. 10.1145/2372233.2372235.

Understanding the algorithm from the nature of the problem is the main advantage of this paradigm over the imperative paradigm, since it is easier and more feasible to build models, experiment and expose. It is a notion that should be mainstream in the IT industry, if we think not to downplay the importance of secure, predictable, and efficient systems. However, there is a small part of programmers who think about it and use the functions paradigm on functional languages to design, model and test their algorithms, and then transcend them to imperative language.

The misconception about this paradigm in the IT industry, perhaps it is fueled by Universities, who use this type of paradigm only to teach an introduction to programming on the design and analysis of algorithms. Coining ideas that functional programming languages only serve to teach fundamental concepts about algorithms and not as a standard to follow that guarantees to truly understand the nature of the problem and then carry out implementations for production. Among other ideas such as a strict theoretical knowledge of lambda calculus is necessary, or that the students cannot understand abstract concepts[1].

Lambda calculus is a powerful way to define functions of superior type using clean technical properties. In the lambda calculus untyped, everything is a function and a closed term in a normal form can only be an abstraction λx.A., It is computationally complete, being able to represent natural numbers, lists, booleans, calculation functions of addition, successor, predecessor, subtraction, multiplication, and others.

The typed lambda allows the use of constants that represent various data types and primitive functions. For example, natural numbers with more, times, etc. [2] The function can be applied only if it can accept the data type in its input, however it maintains the power of the lambda calculation.

Algebra n - 1

Function Definition
(if n $==0$) 0 *else (n* - 1)

Lambda Expression,
PRED := $\lambda n.\lambda f.\lambda x.$n $(\lambda g.\lambda h.h(g\ f))$ $(\lambda u.x)$ $(\lambda u.u\ )$

Piecewise,
$$\text{pred}(n) = \begin{cases} 0 & if\ n = 0, \\ n-1 & otherwise \end{cases}$$

Haskell,
```
pred :: Int ⇒ Int
pred n | n == 0 = 0
       | otherwise = n - 1
```

Here a sample of how a predecessor function can be represented in different notations, from algebraic notation, untyped lambda calculus and finally a functional language like Haskell.

4 Isabelle/HOL Proof Assistant

Isabelle dates back to 1986. Its development in subsequent years included a higher order meta-logic, added improvements in the use of quantifiers, improved understanding of natural deduction, polymorphism, and an analyzer was added among other tools. In the 90s, the possibility of declaring type classes was added, important for many types of high-order logic. The 1994 version introduced more support for different theories[3].

Isabelle is a proof assistant that implements several theory libraries, such as Higher Order Logic (HOL) usually the most used, First Order Logic (FOL), Zermelo-Fraenkel Set Theory(ZF), Classical Computational Logic (CLL), Logic for Computable Functions (LFC), Classical First-Order Logic with Proofs (FOLP), Constructive Type Theory (CTT), and other miscellaneous.

Isabelle / HOL is characterized by its ability to define functions in a style similar to a functional paradigm programming language such as Haskell or Standard ML.

```
fun conj :: "bool ⇒ bool ⇒ bool" where
  "conj True True = True" |
  "conj _ _ = False"
```

Isabelle has base types, *bool* for truth values, *nat* for the natural number type $\mathbb{N}$, and *int* for the mathematical integer type $\mathbb{Z}$. As a first approach to Isabelle's syntax, the *value* command allows us to evaluate expressions and functions in a direct way.

value "suc 0"

The result of this expression can be viewed in the Isabelle user interface output tab, as "1" :: "nat", Isabelle has the ability to infer types as long as there is no uncertainty in the expression, for example.

value "1 + 1"

In this case the program cannot derive the type, because the type of the first operand is unknown, which triggers a "Wellsortedness error", so we must specify whether it is a *nat* or an *int*.

value "(1 :: int) + 1"

Isabelle has the *main* theory, which stores all the main predefined types, their functions and operators. You can see their content in the integrated documentation tab. Regarding special infix operators, we can place them without the need for quotation marks (``).

value "(6::nat) div 3"

A basic code scheme in Isabelle is composed of the name of the theory, the word *import* to bring the parent theories, the word *begin* to indicate where the content of the theory begins, from where the types, functions, theorems and proofs contained in the theory are declared, and the word *end* to indicate where the content ends.

```
theory theoryName
  imports Main
begin
 definitions, theorems and proofs
end
```

In this case, the *Main* theory is imported to make visible the predefined theories, such as arithmetic, lists, sets and others. Each theory $T_1 ... T_n$ is contained in files with the ending *T.thy* Let's consider the following example in Haskell to illustrate the syntactic difference with Isabelle, and learn a way to find bugs in a Haskell algorithm, using lemmas and nitpick command to find counterexamples. In an auditing company, a computer program has the function of randomly choosing 3 auditors to audit a contract. But the program must first know if they are available or not. So the function receives a status and a list of auditors, and generates a new list, without the auditors that have the status that we passed as an argument.

Haskell Code

```
data Auditor = Auditor { firstName :: String
                       , secondName :: String
                       , currentStatus :: Status
                       } deriving (Eq, Show)

data Status = Busy | Available deriving (Eq, Show)

removAuditor :: Status -> [Auditor] -> [Auditor]
removAuditor x [] = []
removAuditor x (y:ys) | x == knowCurrentStatus y = ys
                      | otherwise = (y : removAuditor x
                      ys)

knowCurrentStatus :: Auditor -> Status
knowCurrentStatus x = currentStatus x
```

In Haskell you can call a function that has been declared in a later line of code. In Isabelle we must respect the order in which functions and types are declared. So we can notice the datatype *status*, it is before the other datatype *auditor*, which contains the identifier currentStatus with the *status* data.

```
theory Auditors
  imports Main
begin

datatype status = busy | available

datatype auditor = auditor (firstName: string)
                           (secondName: string)
                           (currentStatus: status)

definition knowCurrentStatus :: “auditor ⇒ status” where
        “knowCurrentStatus x = currentStatus x”

fun removAuditor :: “status ⇒ auditor list ⇒ auditor list” where
   “removAuditor x [] = []” |
   “removAuditor x (y#ys) = (if x = knowCurrentStatus y
                             then ys
                             else y#(removAuditor x ys))”
end
```

The first notable difference with Haskell, is that the types are written in lowercase. There are several ways to define datatypes and their recursive forms. There are also several ways to define functions, but it will be explained in depth later. The use of the quotation marks (“ ”) is important for Isabelle to interpret the syntax that is inside, We can also use the *value* command to observe the sequence of types that a function contains, in the output tab.

Isabelle

```
value “removAuditor”
“_”:: “status ⇒ auditor list ⇒ auditor list”
```

In principle, this function must receive a *status*. For this specific case a *busy* status, and it also receives a list of auditors.

Isabelle

```
value “removAuditor busy [
(auditor “Lawrence” “Muggerud” available),
(auditor “Mike” “Konan” busy)
] ”
```

```
“[auditor “Lawrence” “Muggerud” available]” :: ”auditor list”
```

In effect, the program returns a new list only with the auditor available, however we do not know if there is another type of behavior, so we need to find counter-examples. If this function should return a new list with available auditors, then we can create a function that determines if this is true or false, which will help us find the counterexamples.

```
fun contains :: “auditor list ⇒ status ⇒ bool” where
  “contains [] _ = False” |
  “contains (x#xs) y = (if knowCurrentStatus x = y
                         then True
                         else contains xs y)”
```

Isabelle

```
lemma “¬ (contains (removAuditor a b) a)”
    nitpick
```

```
proof (prove)
goal (1 subgoal):
1.¬ contains (removAuditor a b) a
Auto Quickcheck found a counterexample:
  a = busy
  b = [auditor [] [] busy, auditor [] [] busy]
```

The Auto Quickcheck has found that by making use of free variables such as *a* (to refer to the type of status) and *b* (list of auditors), there is a counterexample. An unwanted behavior that we have to correct. Auto Quickcheck may have shown a counterexample, otherwise nitpick command can be used. We can see that variable *b* contains two auditors in status *busy*. That is, two auditors with the same status. So we proceed to carry out this example to observe their behavior.

Isabelle

```
value “removAuditor busy [
(auditor “Mike” “Konan” busy)
(auditor “Jennifer” “Connelly” busy)
] ”
```

```
“[auditor “Jennifer” “Connelly” busy]” :: ”auditor list”
```

So we have found a bug, an unwanted behavior of our algorithm that we have to correct, as only one of the busy auditors was removed, and not all of them.

The lemma command is used to declare auxiliary propositions or auxiliary theorems before the final conclusion in the middle of a proof. Below lemma we can to apply the proof commands, such as the nitpick command. This helps us find counterexamples, which are exceptions to the generalization. Ideally, a solid algorithm must have a general rule and no exceptions. Our algorithm must generate a new list of auditors without the auditors with the status that we passed as an argument for all the cases. However, the bug shows that it does not comply with this. When there is more than one auditor with the same status, this is the exception that we must eliminate.

```
fun removAuditor :: “status ⇒ auditor list ⇒ auditor list” where
  “removAuditor x [] = []” |
  “removAuditor x (y#ys) = (if x = knowCurrentStatus y
                              then removAuditor x ys
                              else y#(removAuditor x ys))”
```

The call in blue will allow that when the condition is true, the same function is called recursively, as many times as necessary until the length of the list of auditors is decreased. This will allow creating a new list of auditors without the status that we want, without exceptions.

Isabelle

```
lemma “¬ (contains (removAuditor a b) a)”
  nitpick
```

```
Nitpick found no counter example
```

```
Isabelle
value "removAuditor busy [
(auditor "Mike" "Konan" busy)
(auditor "Jennifer" "Connelly" busy)
(auditor "Lawrence" "Muggerud" available)
] "
```

```
"[auditor "Lawrence" "Muggerud" available]" :: "auditor list"
```

Once the function is corrected, the result is as expected, however we must verify the numerical integrity of the length of the previous list with the new list, so we proceed to place another property. As we previously started a proof state with the first lemma, even if it is true, we have to tell Isabelle that we want to place another lemma and that the proof has not yet been completed. So we must place the *sorry* command and then declare the following lemma.

```
Isabelle
lemma "(auditorLength b a) = (auditorLength(removAuditor a b) a)
+ (auditorLength b a)"
    nitpick
```

```
Nitpick found no counter example
```

With this lemma we define that the initial number of auditors with status *a* contained in the original list, once it passes through the *removAuditor* function, is 0 which verifies that the same number of auditors with status a were discarded in their entirety. The function *auditorLength* returns natural numbers and is conformed as follows.

```
fun lengthAcc :: "auditor list ⇒ status ⇒ nat ⇒ nat" where
    "lengthAcc [] _ n = n" |
    "lengthAcc (x#xs) y n = (if knowCurrentStatus x = y
                              then lengthAcc xs y (n+1)
                              else lengthAcc xs y n)"

fun auditorLength :: "auditor list ⇒ status ⇒ nat" where
    "auditorLength xs y = lengthAcc xs y 0"
```

Once we have corrected all the bugs and eliminated all the counterexamples, we can run a proof.

Isabelle

```
lemma “(auditorLength b a) = (auditorLength(removAuditor a b) a)
+ (auditorLength b a)”
    apply(induct b)
    apply(simp)
    apply(auto)
    done
```

```
proof (prove)
goal (2 subgoals):
1. auditorLength [] a =
auditorLength (removAuditor a []) a + auditorLength [] a
2. ⋀aa b.
auditorLength b a =
auditorLength (removAuditor a b) a + auditorLength b a
auditorLength (aa # b) a =
auditorLength (removAuditor a (aa # b)) a +
auditorLength (aa # b) a
```

When the proof command apply(induct b) is placed, two cases or subgoals appear which we must test. These are shown in the output tab, when clicking the apply command. In this case it shows that the first subgoal considers a list of empty auditors, which we can prove by means of simplification (simp).

```
proof (prove)
goal (1 subgoal):
1. ⋀aa b.
auditorLength b a =
auditorLength (removAuditor a b) a + auditorLength b a
auditorLength (aa # b) a =
auditorLength (removAuditor a (aa # b)) a +
auditorLength (aa # b) a
```

There is only one subgoal to prove, the apply(auto) command will be in charge of testing it automatically. It will test with all the rules that Isabelle contains, the auto command also applies (simp), but sometimes it is necessary to place it individually to add special rules within parentheses.

```
proof (prove)
goal:
No subgoals!
```

All the subgoals are already tested, and they are true, however if we want to test another lemma after having applied an auto command, we will use **by auto** instead. Let's see the complete example.

```
theory Auditors
  imports Main
begin

datatype status = busy | available

datatype auditor = auditor (firstName: string)
                          (secondName: string)
                          (currentStatus: status)

definition knowCurrentStatus :: "auditor ⇒ status" where
        "knowCurrentStatus x = currentStatus x"

fun removAuditor :: "status ⇒ auditor list ⇒ auditor list" where
   "removAuditor x [] = []" |
   "removAuditor x (y#ys) = (if x = knowCurrentStatus y
                              then removAuditor x ys
                              else y#(removAuditor x ys))"

fun contains :: "auditor list ⇒ status ⇒ bool" where
   "contains [] _ = False" |
   "contains (x#xs) y = (if knowCurrentStatus x = y
                         then True
                         else contains xs y)"

fun lengthAcc :: "auditor list ⇒ status ⇒ nat ⇒ nat" where
   "lengthAcc [] _ n = n" |
   "lengthAcc (x#xs) y n = (if knowCurrentStatus x = y
                            then lengthAcc xs y (n+1)
                            else lengthAcc xs y n)"

fun auditorLength :: "auditor list ⇒ status ⇒ nat" where
   "auditorLength xs y = lengthAcc xs y 0"

lemma first_lemma:
"¬(contains(removAuditor a b) a)"
 nitpick
 sorry

lemma second_lemma:
"(auditorLength b a) = (auditorLength(removAuditor a b)a) + (auditorLength b a)"
 apply (induct b)
 apply simp
 by auto

lemma last_lemma:
"(removAuditor a b) = (removAuditor a b) → ¬ (contains (removAuditor a b) a)"
 apply(induct b)
 apply simp
```

proof –
 fix aa :: *auditor* **and** ba :: "*auditor list*"
 assume "*removAuditor* a ba = *removAuditor* a ba → ¬ *contains (removAuditor* a ba) a"
 then have "¬ *contains (removAuditor* a ba) a"
  by meson
 then show "*removAuditor* a (aa # ba) = *removAuditor* a (aa # ba) →
¬ *contains (removAuditor* a (aa # ba)) a"
  using *contains*.simps(2) *removAuditor*.simps(2) by presburger
qed
end

Isabelle contains a tool called *sledgehammer*, which is used to search for theorems, proofs and methods on external servers, which could help to test our lemmas. The last **proof** command was added by Isabelle's external database, through the sledehammer tool, we can finally see how this proof solves the last subgoals, and generates the theorem in the output tab.

```
theorem
last_lemma: removAuditor ?a ?b = removAuditor ?a ?b →
¬ contains (removAuditor ?a ?b) ?a
```

Although isabelle is a general purpose tool, it adapts very well to the syntax of other functional paradigm languages, in this way the computational scientist familiar with languages such as Haskell, scala and others, will find a faster adaptability, interpretation and formal understanding of their theories with the help of natural language and automated tests. This is a typed version of the exercise [4] which does not use Isabelle's internal libraries like (List), in order to learn about the nature of the problem and its solution.

References

[1] Vujosevic Janicic, Milena and Tošić, Dušan. The role of programming paradigms in the first programming courses. *The Teaching of Mathematics. 2008.*

[2] D. A. Turner. Some History of Functional Programming Languages. *University of Kent and Middlesex University.* 2012.

[3] Laurence C. Paulson. Old Introduction to isabelle. With Tobias Nipkow and Markus Wenzel contributions. *Computer Laboratory, University of cambridge.* 2021.

[4] Thomas Genet. A Short Isabelle/HOL Tutorial for the Functional Programmer. [Research Report] IRISA. 2017. hal-01208577v5